NASA Contractor Report 191587
ICASE Report No. 93-99

ICASE

RUNTIME SUPPORT AND COMPILATION METHODS FOR USER-SPECIFIED DATA DISTRIBUTIONS

Ravi Ponnusamy
Joel Saltz
Alok Choudhury
Yuan-Shin Hwang
Geoffrey Fox

NASA Contract No. NAS1-19480
December 1993

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23681-0001

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-0001

Runtime Support and Compilation Methods for User-Specified Data Distributions*

Ravi Ponnusamy[†‡] **Joel Saltz**[†] **Alok Choudhary**[‡] **Yuan-Shin Hwang**[†]
Geoffrey Fox[‡]

[†]*UMIACS and Computer Science Dept.*
University of Maryland
College Park, MD 20742

[‡]*Northeast Parallel Architectures Center*
Syracuse University
Syracuse, NY 13244

Abstract

This paper describes two new ideas by which an HPF compiler can deal with irregular computations effectively. The first mechanism invokes a user specified mapping procedure via a set of compiler directives. The directives allow use of program arrays to describe graph connectivity, spatial location of array elements and computational load. The second mechanism is a simple conservative method that in many cases enables a compiler to recognize that it is possible to reuse previously computed information from inspectors (e.g. communication schedules, loop iteration partitions, information that associates off-processor data copies with on-processor buffer locations). We present performance results for these mechanisms from a Fortran 90D compiler implementation.

*This work was sponsored in part by ARPA (NAG-1-1485), NSF (ASC 9213821) and ONR (SC292-1-22913). Also supported by NASA Contract No. NAS1-19480 while author Saltz was in residence at ICASE, NASA Langley Research Center, Hampton, Virginia. Author Choudhary was also supported by NSF Young Investigator award (CCR-9357840). The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

1 Introduction

1.1 Background

We address a class of irregular problems that consists of a sequence of clearly demarcated concurrent computational phases where patterns of data access and computational cost cannot be anticipated until runtime. In this class of problems, once runtime information is available, data access patterns are known before each computational phase. We call these problem *irregular concurrent problems* [10]. Examples of irregular concurrent problems include adaptive and self-adaptive explicit, multigrid unstructured computational fluid dynamic solvers [31, 38, 16], molecular dynamics codes (CHARMM, AMBER, GROMOS, etc.) [6], diagonal or polynomial preconditioned iterative linear solvers [39], and time dependent flame modeling codes [34].

In this paper, we focus on the runtime support, the language extensions and the compiler support required to provide efficient data and work load distribution. We present methods and a prototype implementation that make it possible for compilers to efficiently handle irregular problems coded using a set of language extensions closely related to Fortran D [15], Vienna Fortran [43] and High-Performance Fortran (HPF).

The optimizations that must be carried out to solve irregular concurrent problems efficiently on a distributed memory machine include:

1. data partitioning
2. partitioning computational work
3. software caching methods to reduce communication volume
4. communication vectorization to reduce communication startup costs

Since data access patterns are not known in advance, decisions about data structure and workload partitioning have to be deferred until runtime. Once data and work have been partitioned between processors, prior knowledge of loop data access patterns makes it possible to predict which data needs to be communicated between processors. This ability to predict communication requirements makes it possible to carry out communication optimizations. In many cases, communication volume can be reduced by pre-fetching only a single copy of each referenced off-processor datum. The number of messages can also be reduced by using data access pattern knowledge to allow pre-fetching quantities of off-processor data. These two optimizations are called *software caching* and *communication vectorization*.

Whenever there is a possibility that a loop's data access patterns might have changed between consecutive loop invocations, it is necessary to repeat the preprocessing needed to minimize communication volume and startup costs. When data access patterns change, it may also be necessary to repartition computational work. Fortunately, in many irregular concurrent problems, data access patterns change relatively infrequently. In this paper, we present simple conservative techniques that in many cases make it possible for a compiler to verify that data access patterns remain unchanged between loop invocations, making it possible to amortize the associated costs of software caching and message vectorization.

Figure 1 illustrates a simple sequential Fortran irregular loop (loop L2) which is similar in form to loops found in unstructured computational fluid dynamics (CFD) codes and molecular dynamics codes In Figure 1, arrays **x**

```
C Outer loop L1
   do i = 1, n_step
   ...
C Inner Loop L2
   do i=1, nedge
     y(edge1(i)) = y(edge1(i)) + f(x(edge1(i)), x(edge2(i)))
     y(edge2(i)) = y(edge2(i)) + g(x(edge1(i)), x(edge2(i)))
   end do
   ...
   end do
```

Figure 1: An Example code with an Irregular Loop

and **y** are accessed by indirection arrays **edge1** and **edge2**. Note that the data access pattern associated with the inner loop, loop L2 is determined by integer arrays **edge1** and **edge2**. Because arrays **edge1** and **edge2** are not modified within loop L2, L2's data access pattern can be anticipated prior to executing L2. Consequently, **edge1** and **edge2** are used to carry out preprocessing needed to minimize communication volume and startups. Whenever it can be determined that **edge1, edge2**, and **nedge** have not been modified between consecutive iterations of outer loop L1, repeated preprocessing can be avoided.

1.2 Irregular Data Distribution

On distributed memory machines, large data arrays need to be partitioned between local processor memories. These partitioned data arrays are called *distributed arrays*. Long term storage of distributed array data is assigned to specific processor and memory locations in the machine. Many applications can be efficiently implemented by using simple schemes for mapping distributed arrays. One example of such a scheme would be the division of an array into equal sized contiguous subarrays and assignment of each subarray to a different processor. Another example would be to assign consecutively indexed array elements to processors in a round-robin fashion. These two data distribution schemes are often called BLOCK and CYCLIC data distributions [14], respectively.

Researchers have developed a variety of heuristic methods to obtain data mappings that are designed to optimize irregular problem communication requirements [37, 41, 29, 27, 3, 20]. The distribution produced by these methods typically results in a table that lists a processor assignment for each array element. This kind of distribution is often called an *irregular distribution*.

Partitioners typically make use of one or more of the following types of information:

1. a description of graph connectivity

2. spatial location of array elements

3. information that associates array elements with computational load

Languages such as High Performance Fortran, Fortran D and Vienna Fortran allow users to advise the compiler on how array elements should be assigned to processor memories. In HPF a pattern of data mapping can be specified using the **DISTRIBUTE** directive. Two major types of patterns can be specified this way: **BLOCK** and **CYCLIC** distributions. For example,

REAL, DIMENSION(500,500) :: X, Y

!HPF $ DISTRIBUTE (*, BLOCK) :: X

!HPF $ DISTRIBUTE (BLOCK, BLOCK) :: Y

breaks the arrays **X** and **Y** into groups of columns and rectangular blocks, respectively.

In this paper, we describe an approach where the user does not *explicitly* specify a data distribution. Instead the user specifies:

1. the type of information to be used in data partitioning, and

2. the irregular data partitioning heuristic to be used

We have designed and implemented language extensions to allow users to specify the information needed to produce an irregular distribution. Based on user directives, the compiler produces code that, at runtime, passes the user specified partitioning information to a (user specified) partitioner.

To our knowledge, the implementation described in this paper was the first distributed memory compiler to provide this kind of support. User specified partitioning has recently been implemented in the D System Fortran 77D compiler [19]; the CHAOS runtime support described in this paper has been employed in this implementation. In the Vienna Fortran [43] language definition a user can specify a customized distribution function. The runtime support and compiler transformation strategies described here can also be applied to Vienna Fortran.

We have implemented our ideas using the Syracuse Fortran 90D/HPF compiler [5]. We have made the following assumptions:

1. irregular accesses are carried out in the context of a single or multiple statement parallel loop where dependence between iterations may occur due to reduction operations only (e.g. addition, max, min, etc.), and

2. irregular array accesses occur as a result of a single level of indirection with a distributed array that is indexed directly by the loop variable

1.3 Organization

This paper is organized as follows. The context of the work is outlined in Section 2. Section 3 describes the runtime technique that saves and reuses results from previously performed loop pre-processing. Section 4

describes the data structure, the compiler transformations, and the language extensions used to control compiler-linked runtime partitioning. Section 5 presents the runtime support developed for coupling data partitioners, for partitioning workload and for managing irregular data distributions. Section 6 presents data to characterize the performance of our methods. Section 7 provides a summary of related work, and Section 8 concludes.

2 Overview

2.1 Problem Partitioning and Application Codes

It is useful to describe application codes to introduce the motivation behind preprocessing. We first describe two application codes (an unstructured Euler solver and a molecular dynamics code) that consist of a sequence of loops with indirectly accessed arrays; these are loops analogous to those depicted in Figure 1. We then describe a combustion code with a regular data access pattern but with highly non-uniform computational costs. In that code, computational costs vary dynamically and cannot be estimated until runtime.

2.1.1 Codes with Indirectly Accessed Arrays

The first application code is an unstructured Euler solver used to study the flow of air over an airfoil [31, 38, 23]. Complex aerodynamic shapes require high resolution meshes and, consequently, large numbers of mesh points. A mesh vertex is an abstraction represented by Fortran array data structures. Physical values (e.g. velocity, pressure) are associated with each mesh vertex. These values are called *flow variables* and are stored in arrays. Calculations are carried out using loops over the list of edges that define the connectivity of the vertices. For instance, Figure 1 sweeps over *nedges* mesh edges. Loop iteration i carries out a computation involving the edge that connects vertices $edge1(i)$ and $edge2(i)$.

To parallelize an unstructured Euler solver, one needs to partition mesh vertices (i.e. arrays that store flow variables). Since meshes are typically associated with physical objects, a spatial location can often be associated with each mesh point. The spatial location of the mesh points and the connectivity of the vertices is determined by the mesh generation strategy [40, 30]. Figure 2 depicts a mesh generated by such a process. This is an unstructured mesh representation of a three dimensional aircraft wing.

The way in which the vertices of such irregular computational mesh are numbered frequently does not have a useful correspondence to the connectivity pattern (edges) of the mesh. We partition mesh points to minimize communication. Recently promising heuristics have been developed that can use one or several of the following types of information: 1) spatial locations of mesh vertices, 2) connectivity of the vertices, and 3) estimates of the computational load associated with each mesh point. For instance, a user might choose a partitioner that is based on coordinates [3] to partition data. A coordinate bisection partitioner decomposes data using the spatial location of vertices in the mesh. If the user chooses a graph based partitioner, such as the spectral partitioner [37], the connectivity of the mesh could be used to decompose the data.

The next step in parallelizing this application involves assigning equal amount of work to processors. An Euler solver consists of a sequence of loops that sweep over a mesh. Computational work associated with each loop must be partitioned between processors to balance load. Our approach is to assign all work associated with a given loop iteration to a single processor. Consider a loop that sweeps over mesh edges, closely resembling

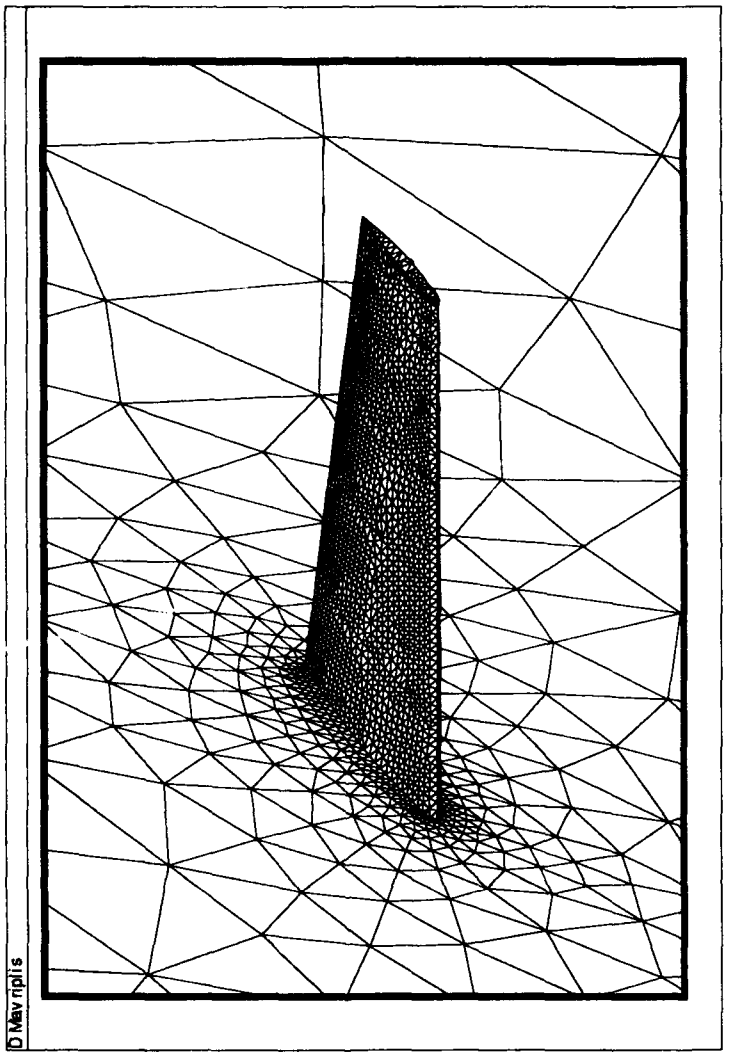

Figure 2: An Example Unstructured Mesh

Table 1: Application Area Specific Terminology

Program Representation	Unstructured Mesh	Molecular Dynamics
Data Array Elements	Physical State for each Mesh Vertex	Force Components for Each Atom
Loop Iterations	Mesh Edges	Partner list
Array Distribution	Partition of Mesh Vertices	Partition of Atoms
Loop Iteration Partition	Edge Partition	Partition of Non-bonded Force Calculations

the loop depicted in Figure 1. We would partition mesh edges so that 1) we obtain a good load balance and 2) computations mostly employ locally stored data.

Other unstructured problems have analogous indirectly accessed arrays. For instance, consider the non-bonded force calculation in the molecular dynamics code CHARMM [6]. Figure 5 depicts the non-bonded force calculation loop. Force components associated with each atom are stored as Fortran arrays. The outer loop L1 sweeps over all atoms; in this discussion, we assume that L1 is a parallel loop. Each iteration of L1 is carried out on a single processor, so loop L2 need not be parallelized.

All atoms within a given cutoff radius interact with each other. The array **Partners(i, *)** list all the atoms that interact with atom i. Inside the inner loop, the three force components (x,y,z) between atom i and atom j are calculated (Vander Waal's and electrostatic forces). They are then added to the forces associated with the atom i and subtracted from the forces associated with the atom j.

We attempt to partition force array elements to reduce interprocessor communication in the non-bonded force calculation loop (Figure 5). Figure 3 depicts a possible distribution of force array elements to four processors. This figure depicts a Myoglobin molecule in which shading is used to represent the assignment of atoms to processors. Data sets associated with sequential versions of CHARMM associate each atom with an arbitrary index number. We depict a distribution that assigns consecutively numbered sets of atoms to each processor (i.e. a **BLOCK** distribution). Since nearby atoms interact, we see that in this case, the choice of a **BLOCK** distribution is likely to result in a large volume of communication. Consider instead a distribution based on the spatial locations of atoms. Figure 4 depicts a distribution of atoms to processors carried out using a coordinate bisection partitioner [3]. When we compare Figure 3 with Figure 4, we see that the later figure has a much smaller amounts of surface area between the portions of the molecule associated with each processor.

Table 1 summarizes the application area specific terminology used to describe data array elements, loop iterations, array distributions and loop iteration partition.

2.1.2 A Code with Time Varying Computational Costs

We now describe a type of application code that is qualitatively different from the unstructured Euler and molecular dynamics codes previously discussed. This type of code is used to carry out detailed time dependent,

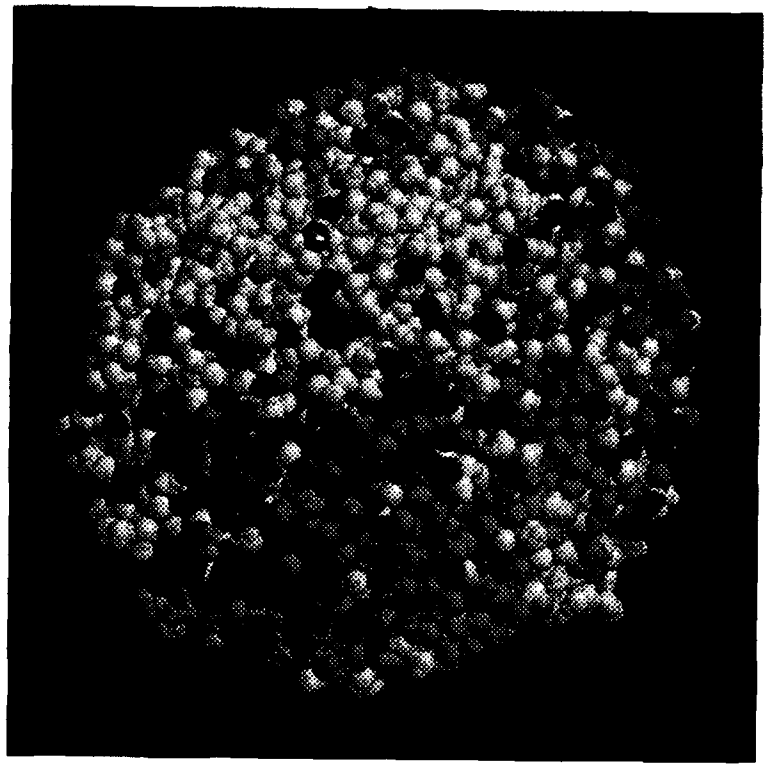

Figure 3: BLOCK Distribution of Atoms

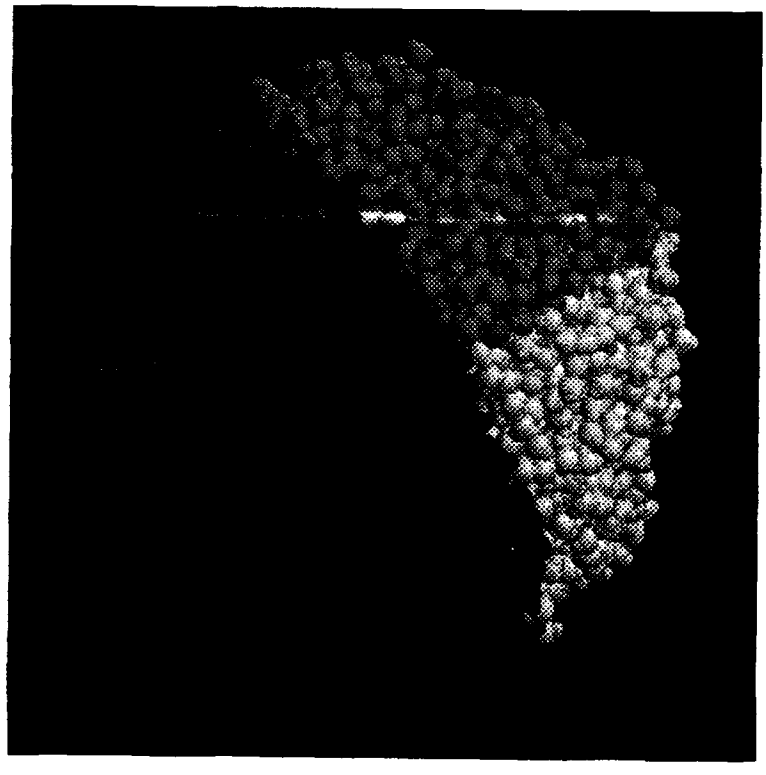

Figure 4: Distribution of Atoms Using a Partitioner

```
L1: do i = 1, NATOM
    L2: do index = 1, INB(i)
            j = Partners(i, index)
            Calculate dF (x, y and z components).
            Subtract dF from F_j.
            Add dF to F_i
        end do
    end do
```

Figure 5: Non-bonded Force Calculation Loop from CHARMM

multi-dimensional flame simulations. The calculation cycles between two distinct phases. The first phase (*convection*) calculates fluid convection over a Cartesian mesh. The second phase (*reaction*) solves the ordinary differential equations used to represent chemical reactions and energy release. During the reaction phase, a set of local computations are carried out at each mesh point. The computational costs associated with the reaction phase varies from mesh point to mesh point since at each mesh point an adaptive method is used to solve the system of ordinary differential equations. Arrays in this application are not indirectly accessed as in the previous two example applications.

In Figure 6 we present a simplified one dimensional version of this code. The convection phase (loop nest L2) consists of a sweep over a structured mesh involving array elements located at nearest neighbor mesh points. The reaction phase (loop nest L3) involves only local calculations. The computational cost associated with the function *Adaptive_Solver* depends on the value of x(i). It is clear that the cost of *Adaptive_Solver* can vary from mesh point to mesh point. The cost of *Adaptive_Solver* at a given mesh point that slowly changes between iterations of the outer loop L1.

There are a number of strategies that can be used in partitioning data and work associated with this flame code. If the convection calculations comprise the bulk of the computation time, it would be reasonable to partition the mesh (arrays **x,y** and **z** in Figure 6) into equal sized blocks.

However, the reaction calculations (loop nest L3 in Figure 6) usually comprise at least half of the total computational cost. A majority of the work associated with the reaction phase of the calculation is carried out on a small fraction of the mesh points. Our current approach involves maintaining a block mapping of the mesh (arrays **x,y** and **z**) during the convection phase. (In actual codes, we typically deal with two or three dimensional meshes represented as two or three dimensional arrays). In order to ensure a good load balance during the reaction phase, we redistribute only expensive reaction calculations. In Figure 6, we must transmit array element $x(i)$ in order to redistribute the reaction calculation for mesh point i. Once the reaction calculation is carried out, the solution $z(i)$ is returned to the processor to which it is assigned. At a given mesh point, the cost associated with a reaction calculation generally vary gradually as a problem progresses. This

```
L1: do time=1,timesteps
  C Convection Phase:
    L2: do i = 1, NPOINTS
          x(i) = x(i) + F(y(i),y(i-1), y(i), y(i+1), z(i))
        end do
        y(1:NPOINTS) = x(1:NPOINTS)
  C Reaction Phase:
    L3: do i = 1, NPOINTS
          z(i) = Adaptive_Solver(x(i)
        end do
    end do
```

Figure 6: Overview - Combustion Code

property provides a way to estimate reaction calculation costs in the subsequent computation step.

2.2 Solving Irregular Problems

In this section, we describe how we solve irregular problems efficiently on distributed memory machines. On distributed memory machines the data and the computational work must be divided between individual processors. The criteria for partitioning are minimizing the volume of interprocessor data communication and good load-balancing.

Once distributed arrays have been partitioned, each processor ends up with a set of globally indexed distributed array elements. Each element in a size N distributed array, A, is assigned to a particular home processor. In order for another processor to be able to access a given element, $A(i)$, of the distributed array the home processor and local address of $A(i)$ must be determined. A *translation table* is built that for each array element, lists the home processor and the local address.

Memory considerations make it clear that it is not always feasible to place a copy of the translation table on each processor, so the translation table must be distributed between processors. This is accomplished by distributing the the translation table by blocks i.e. putting the first N/P elements on the first processor, the second N/P elements on the second processor, etc., where P is the number of processors. When an element $A(m)$ of distributed array A is accessed, the home processor and local offset are found in the portion of the distributed translation table stored in processor $((m-1)/N) * P + 1$. We refer to a translation table lookup aimed at discovering the home processor and the offset associated with a global distributed array index as a *dereference request.*

Consider the irregular loop L2 in Figure 1 that sweeps over the edges of a mesh. In this case, distributing data arrays **x** and **y** corresponds to partitioning the mesh vertices; partitioning loop iterations corresponds to partitioning edges of the mesh. Hence, each processor gets a subset of loop iterations (edges) to work on. An edge i that has both end points ($edge1(i)$ and $edge2(i)$) inside the same partition (processor) requires no outside information. On the other hand, edges which cross partition boundaries require data from other processors. Before executing the computation for such an edge, processors must retrieve the required data from other processors.

There is typically a non-trivial communication latency, or message startup cost, in distributed memory machines. We vectorize communication to reduce the effect of communication latency and carry out software caching to reduce communication volume. To carry out either optimization, it is extremely helpful to have a-priori knowledge of data access patterns. In irregular problems, it is generally not possible to predict data access patterns at compile time. For example, the values of indirection arrays **edge1** and **edge2** of loop L2 in Figure 1 are known only at runtime because they depend on the input mesh. During program execution, we pre-process the data references of distributed arrays. On each processor, we pre-compute which data need to be exchanged. The result of this pre-processing is a *communication schedule* [32].

Each processor uses communication schedules to exchange required data before and after executing a loop. The same schedules can be used repeatedly, as long as the data reference patterns remain unchanged. In Figure 1, loop L2 is carried out many times inside loop L1. As long as the indirection arrays **edge1** and **edge2** are not modified within L1, it is possible to reuse communication schedules for L2. We discuss schedule reuse in detail in the Section 3.

2.3 Communication Vectorization and Software Caching

We describe the process of generating and using schedules to carry out communication vectorization and software caching with the help of the example shown in Figure 1. The arrays **x**, **y**, **edge1** and **edge2** are partitioned between the processors of the distributed memory machine. We assume that arrays **x** and **y** are distributed in the same fashion. Array distributions are stored in a distributed translation table. These local indirection arrays are passed to the procedure *localize* as shown in statement S1 in Figure 7.

Figure 7 contains the pre-processing code for the simple irregular loop L2 shown in Figure 1. In this loop, values of array **y** are updated using the values stored in array **x**. Hence, a processor may need an off-processor array element of **x** to update an element of **y** and it may update an off-processor array element of **y**. Our goal is to compute 1) a *gather schedule* – a communication schedule that can be used for fetching off-processor elements of **x**, and 2) a *scatter schedule* – a communication schedule that can be used to send updated off-processor elements of **y**. However, the arrays **x** and **y** are referenced in an identical fashion in each iteration of the loop L2, so a single schedule that represents data references of either **x** or **y** can be used for fetching off-processor elements of **x** and sending off-processor elements of **y**.

A sketch of how the procedure *localize* works is shown in Figure 8. We store the globally indexed reference pattern used to access arrays **x** and **y** in the array **part_edge**. The procedure *localize* dereferences and translates **part_edge** so that valid references are generated when the loop is executed. The buffer for each data array immediately follows the on-processor data for that array. For example, the buffer for data array **y** begins at

```
C Create the required schedules (Inspector)
S1 Collect indirection array traces and call Chaos procedure localize to compute schedule
C The actual computation (Executor)
S2 call zero_out_buffer(x(begin_buffer), off_proc)
S3 call gather(x(begin_buffer), x, schedule)
S4 do i=1, n_local_edge
S5   y(local_edge1(i)) = y(local_edge1(i)) + f(x(local_edge1(i)), x(local_edge2(i)))
S6   y(local_edge2(i)) = y(local_edge2(i)) + g(x(local_edge1(i)), x(local_edge2(i)))
S7 end do
S8 call scatter_add(y(begin_buffer), y, schedule)
```

Figure 7: Node Code for Simple Irregular Loop

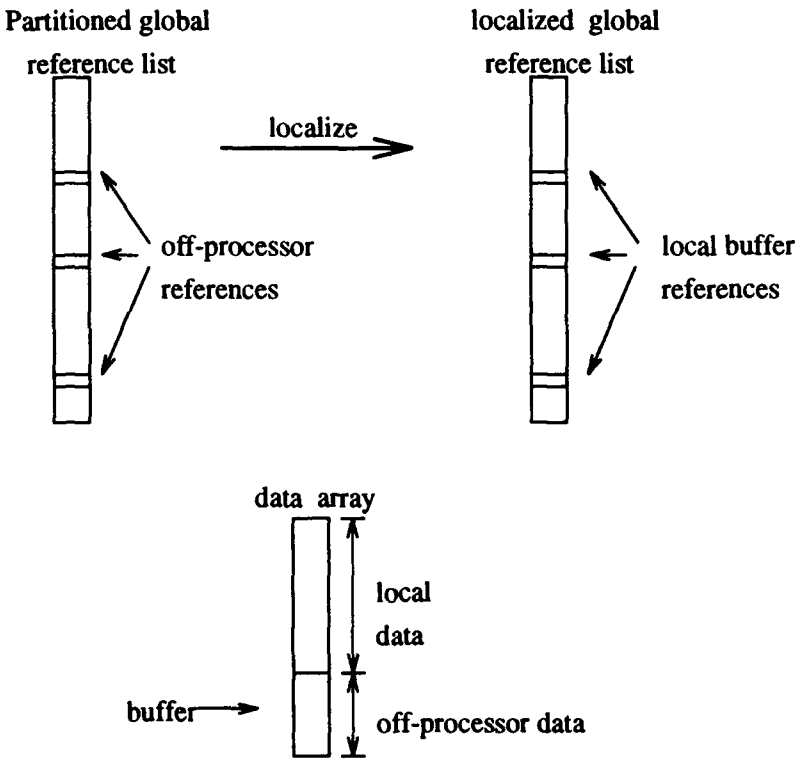

Figure 8: Index Translation by Localize Mechanism

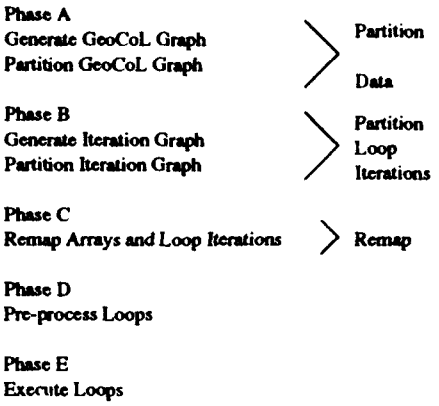

Figure 9: Solving Irregular Problems

y(begin_buffer). Hence, when *localize* translates **part_edge** to **local_edge**, the off-processor references are modified to point to buffer addresses. The procedure *localize* uses a hash table to remove any duplicate references to off-processor elements so that only a single copy of each off-processor datum is transmitted. When the off processor data is collected into the buffer using the schedule returned by *localize*, the data is stored in a way such that execution of the loop using the **local_edge** accesses the correct data.

The executor code starting at S2 in Figure 7 carries out the actual loop computation. In this computation the values stored in the array **y** are updated using the values stored in **x**. During the computation, accumulations to off-processor locations of array **y** are carried out in the buffer associated with array **y**. This makes it necessary to initialize the buffer corresponding to off-processor references of **y**. To perform this action we call the function *zero_out_buffer* shown in statement S2. After the loop computation, the data in the buffer location of array **y** is communicated to the home processors of these data elements (*scatter_add*). There are two potential communication points in the executor code, i.e. the *gather* and the *scatter_add* calls. The *gather* on each processor fetches all the necessary **x** references that reside off-processor. The *scatter_add* calls accumulates the off-processor **y** values. A detailed description of the functionality of these procedures is given in Das et al [12].

2.4 Overview of CHAOS

We have developed efficient runtime support to deal with problems that consist of a sequence of clearly demarcated concurrent computational phases. The project is called CHAOS; the runtime support is called the CHAOS library. The CHAOS library is a superset of the PARTI library [32, 42, 36].

Solving concurrent irregular problems on distributed memory machines using our runtime support involves five major steps (Figure 9). The first three steps in the figure concern mapping data and computations onto processors. We provide a brief description of these steps here, and will discuss them in detail in later sections.

Initially, the distributed arrays are decomposed into a known regular manner.

1. The first step is to decompose the distributed array irregularly with the user provided information. When the user chooses connectivity as one of the information to be used for data partitioning, certain pre-

processing (see Section 5) is required before information can be passed to a partitioner. In Phase A of Figure 9, CHAOS procedures can be called to do the necessary pre-processing. For example, the user may employ a partitioner that uses the connectivity of the mesh shown in Figure 2 or may use a partitioner that uses the spatial information of the mesh vertices. The partitioner calculates how data arrays should be distributed.

2. In Phase B, the newly calculated array distributions are used to decide how loop iterations are to be partitioned among processors. This calculation takes into account the processor assignment of the distributed array elements accessed in each iteration. A loop iteration is assigned to the processor that has the maximum number of local distributed arrays elements accessed in that iteration. Once data is distributed, based on the access patterns of each iteration and data distribution, the runtime routines for this step determine on which processor each iteration will be executed.

3. Once new data and loop iteration distributions are determined, Phase C carries out the actual remapping of arrays from the old distribution to the new distribution.

4. In Phase D, the preprocessing needed for software caching, communication vectorization and index translation is carried out. In this phase, a communication schedule is generated that can be used to exchange data among processor.

5. Finally, in Phase E, information from the earlier phases is used to carry out the computation and communication.

CHAOS and PARTI procedures have been used in a variety of applications, including sparse matrix linear solvers, adaptive computational fluid dynamics codes, molecular dynamics codes and a prototype compiler [36] aimed at distributed memory multiprocessors.

2.5 Overview of Existing Language Support

While our data decomposition directives are presented in the context of Fortran D, the same optimizations and analogous language extensions could be used for a wide range of languages and compilers such as Vienna Fortran, pC++, and HPF. Vienna Fortran, Fortran D and HPF provide a rich set of data decomposition specifications. A definition of such language extensions may be found in Fox et al [15], Loveman et al [14], and Chapman et al [8], [9]. Fortran D and HPF require that users explicitly define how data is to be distributed. Vienna Fortran allows users to write procedures to generate user defined distributions. The techniques described in this paper are being adapted to implement user defined distributions in the Vienna Fortran compiler, details of our Vienna Fortran based work will be reported elsewhere.

Fortran D and Vienna Fortran can be used to *explicitly* specify an irregular partition of distributed array elements. In Figure 10, we present an example of such a Fortran D declaration. In Fortran D, one declares a template called a *distribution* that is used to characterize the significant attributes of a distributed array. The distribution fixes the size, dimension and way in which the array is to be partitioned between processors. A distribution is produced using two declarations. The first declaration is **DECOMPOSITION**. Decomposition fixes the name, dimensionality and size of the distributed array template. The second declaration is

```
S1 REAL*8 x(N),y(N)
S2 INTEGER map(N)
S3 DECOMPOSITION reg(N),irreg(N)
S4 DISTRIBUTE reg(block)
S5 ALIGN map with reg
S6 ... set values of map array using some mapping method ..
S7 DISTRIBUTE irreg(map)
S8 ALIGN x,y with irreg
```

Figure 10: Fortran D Irregular Distribution

DISTRIBUTE. Distribute is an executable statement and specifies how a template is to be mapped onto the processors.

Fortran D provides the user with a choice of several regular distributions. In addition, a user can explicitly specify how a distribution is to be mapped onto the processors. A specific array is associated with a distribution using the Fortran D statement **ALIGN**. In statement S3, of Figure 10, two 1D decompositions, each of size **N**, are defined. In statement S4, decomposition **reg** *is partitioned into equal sized blocks, with one block assigned* to each processor. In statement S5, array **map** is aligned with distribution **reg**. Array **map** will be used to specify (in statement S7) how distribution **irreg** is to be partitioned between processors. An irregular distribution is specified using an integer array; when *map*(i) is set equal to p, element i of the distribution **irreg** is assigned to processor p.

The difficulty with the declarations depicted in Figure 10 is that *it is not obvious how to partition the irregularly distributed array*. The *map* array that gives the distribution pattern of **irreg** has to be generated separately by running a partitioner (the user may supply the partitioner or use one from a library). The Fortran-D constructs are not rich enough for the user to couple the generation of the **map** array to the program compilation process. While there are a wealth of partitioning heuristics available, coding such partitioners from scratch can represent a significant effort. There is also no standard interface between the partitioners and the application codes. In Section 4, we discuss language extensions and compiler support to interface data partitioners.

Figure 11 shows an irregular Fortran 90D Forall loop that is equivalent to the sequential loop L2 in Figure 1. The loop L1 represents a sweep over the edges of an unstructured mesh. Since the mesh is unstructured, an indirection array has to be used to access the vertices during a loop over the edges. In loop L1, a sweep is carried out over the edges of the mesh and the reference pattern is specified by integer arrays **edge1** and **edge2**. Loop L1 carries out reduction operations. That is, the only type of dependency between different iterations of the loop is the one in which they may produce a value to be accumulated (using an associative and commutative operation) in the same array element. Figure 2 shows an example of an unstructured mesh over which such

```
C Sweep over edges: Loop L1
   FORALL i = 1, nedge
S1 REDUCE (ADD, y(edge1(i)), f(x(edge1(i)), x(edge2(i))))
S2 REDUCE (ADD, y(edge2(i)), g(x(edge1(i)), x(edge2(i))))
   END FORALL
```

Figure 11: Example Irregular Loop in Fortran D

computations will be carried out. For example, the loop L1 represents a sweep over the edges of a mesh in which each mesh vertex is updated using the corresponding values of its neighbors (directly connected through edges). Clearly, each vertex of the mesh is updated as many times as the number of neighboring vertices.

The implementation of the Forall construct in High-Performance Fortran follows copy-in-copy-out semantics – loop carried dependencies are not defined. In our implementation, we define loop carried dependencies that arises due to reduction operations. We specify reduction operations in a Forall construct using the Fortran D "REDUCE" construct. Reduction inside a Forall construct is important for representing computations such as those found in sparse and unstructured problems. This representation also preserves explicit parallelism available in the underlying computations.

3 Communication Schedule Reuse

The cost of carrying out an inspector (phases B, C and D in Figure 9) can be amortized when the information produced by the inspector is computed once and then used repeatedly. The compile time analysis needed to reuse inspector communication schedules is touched upon in [18, 13].

We propose a conservative method that in many cases allows us to reuse the results from inspectors. The results from an inspector for loop L can be reused as long as:

- the distributions of data arrays referenced in loop L have remained unchanged since the last time the inspector was invoked
- there is no possibility that the indirection arrays associated with loop L have been modified since the last inspector invocation, and
- the loop bounds of L have not changed

The compiler generates code that, at runtime, maintains a record of when the statements or array intrinsics of a Fortran 90D loop may have written to a distributed array that is used to indirectly reference another distributed array. In this scheme, each inspector checks this runtime record to see whether any indirection arrays may have been modified since the last time the inspector was invoked.

In this presentation, we assume that we are carrying out an inspector for a Forall loop. We also assume that all indirect array references to any distributed array **y** are of the form **y**(ia(i)) where **ia** is a distributed array and *i* is a loop index associated with the Forall loop.

A data access descriptor (DAD) for a distributed array contains (among other things) the current distribution type of the array (e.g. block, cyclic) and the size of the array. In order to generate correct distributed memory code, whenever the compiler generates code that references a distributed array, the compiler must have access to the array's DAD. In our scheme, we maintain a *global* data structure that keeps track of modifications of *any array* with a given DAD.

We maintain a global variable **n_mod** that represents the cumulative number of Fortran 90D loops, array intrinsics or statements that have modified any distributed array. Note that we are not counting the number of assignments to the distributed array, instead we are counting the number of times the program will execute any block of code that writes to a distributed array[1]. **n_mod** may be viewed as a global time stamp. Each time we modify an array **A** with a given data access descriptor DAD(**A**), we update a global data structure **last_mod** to associate DAD(**A**) with the current value of the global variable **n_mod** (i.e. the current global time stamp). Thus when a loop, array intrinsic or statement modifies **A** we set **last_mod(DAD(A)) = n_mod**. If the array **A** is remapped, it means that DAD(**A**) changes. In this case, we increment **n_mod** and then set **last_mod(DAD(A)) = n_mod**.

The first time an inspector for a Forall loop L is carried out, it must perform all the preprocessing. Assume that L has m data arrays x_L^i, $1 \leq i \leq m$, and n indirection arrays, ind_L^j, $1 \leq j \leq n$. Each time an inspector for L is carried out, we store the following information:

1. DAD(x_L^i) for each unique data array x_L^i , for $1 \leq i \leq m$
2. DAD(ind_L^j) for each unique indirection array ind_L^j, for $1 \leq j \leq n$
3. last_mod(DAD(ind_L^j)), for $1 \leq j \leq n$, and
4. the loop bounds of L

We designate the values of DAD(x_L^i), DAD(ind_L^j) and last_mod(DAD(ind_L^j)) stored by L's inspector as L.DAD(x_L^i), L.DAD(ind_L^j) and L.last_mod(DAD(ind_L^j)).

For a given data array x_L^i and an indirection array ind_L^j in a Forall loop L, we maintain *two sets of data access descriptors*. For instance, we maintain,

1. DAD(x_L^i), the current global data access descriptor associated with x_L^i, and
2. L.DAD(x_L^i), a record of the data access descriptor that was associated with x_L^i when L carried out its previous inspector

For each indirection array ind_L^j, we also maintain two time stamps:

- last_mod(DAD(ind_L^j) is the global time stamp associated with the current data access descriptor of ind_L^j and

[1]Note that a Forall construct or an array construct is an atomic operation from the perspective of language semantics, and therefore, it is sufficient to consider one write per construct rather than one write per element.

- L.last_mod(DAD(ind_L^j)) is the global time stamp of data access descriptor DAD(ind_L^j), last recorded by L's inspector

Once L's inspector has been carried out, the following checks are performed before subsequent executions of L. If any of the following conditions are not met, the inspector must be repeated for L.

1. DAD(x_L^i) == L.DAD(x_L^i), $1 \leq i \leq m$
2. DAD(ind_L^j) == L.DAD(ind_L^j), $1 \leq j \leq n$
3. last_mod(DAD(ind_L^j)) == L.last_mod(L.DAD(ind_L^j)), $1 \leq j \leq n$, and
4. the loop bounds of L remain unchanged

As the above algorithm tracks possible array modifications at runtime, there is potential for high runtime overhead in some cases. The overhead is likely to be small in most computationally intensive data parallel Fortran 90 codes (see Section 6). Calculations in such codes primarily occur in loops or Fortran 90 array intrinsics, so we need to record modifications to a DAD *once* per loop or array intrinsic call.

We employ the same method to track possible changes to arrays used in the construction of the data structure produced at runtime to link partitioners with programs. We call this data structure a GeoCoL graph, and it will be described in Section 4.1.1. This approach makes it simple for our compiler to avoid generating a new GeoCoL graph and carrying out a potentially expensive data repartition when no change has occurred.

We could further optimize our inspector reuse mechanism by noting that there is no need to record modifications to *all* distributed arrays. Instead, we could limit ourselves to recording possible modifications of the sets of arrays that have the same data access descriptor as an indirection array. Such optimization will require inter-procedural analysis to identify the sets of arrays that must be tracked at runtime. Future work will include exploration of this optimization.

4 Coupling Partitioners

In irregular problems, it is often desirable to allocate computational work to processors by assigning all computations that involve a given loop iteration to a single processor [4]. Consequently, we partition *both* distributed arrays and loop iterations using a two-phase approach (Figure 9). In the first phase, termed the "data partitioning" phase, distributed arrays are partitioned. In the second phase, called "workload partitioning", loop iterations are partitioned using the information from the first phase. This appears to be a practical approach, as in many cases the same set of distributed arrays are used by many loops. The following two subsections describe the two phases.

4.1 Data Partitioning

When we partition distributed arrays, we have not yet assigned loop iterations to processors. We assume that loop iterations will be partitioned using a user-defined criterion similar to that used for data partitioning. In the absence of such a criterion, a compiler will choose a loop iteration partitioning scheme; e.g., partitioning

Table 2: Common Partitioning Heuristics

Partitioner	Reference	Spatial Information	Connectivity Information	Vertex Weight	Edge Weight
Spectral Bisection	[37]		√	√	√
Coordinate Bisection	[3]	√		√	
Hierarchical Subbox Decomposition	[11]	√		√	
Simulated Annealing	[29]		√	√	√
Neural Network	[29]		√	√	√
Genetic Algorithms	[29]		√	√	√
Inertial Bisection	[33]	√		√	
Kernighan - Lin	[24]		√	√	√

loops so as to minimize non-local distributed array references. Our approach to data partitioning makes an implicit assumption that most (although not necessarily all) computation will be carried out in the processor associated with the variable appearing on the left hand side of each statement – we call this the *almost owner computes rule*.

There are many partitioning heuristics methods available based on physical phenomena and proximity [37, 3, 41, 20]. Table 2 lists some of the commonly used heuristics and the type of information they use for partitioning. Most data partitioners make use of undirected connectivity graphs and spatial information. Currently these partitioners must be coupled to user programs manually. This manual coupling is particularly troublesome and tedious when we wish to make use of parallelized partitioners. Further, partitioners use different data structures and are very problem dependent, making it extremely difficult to adapt to different (but similar) problems and systems.

4.1.1 Interface Data Structures for Partitioners

We link partitioners to programs by using a data structure that stores information on which data partitioning is to be based. Data partitioners can make use of different kinds of program information. Some partitioners operate on data structures that represent undirected graphs [37, 24, 29]. Graph vertices represent array indices, graph edges represent dependencies. Consider the example loop L1 in Figure 11. The graph vertices represent the N elements of arrays **x** and **y**. The graph edges of the loop in Figure 11 are the union of the edges linking vertices $edge1(i)$ and $edge2(i)$.

In some cases, it is possible to associate geometrical information with a problem. For instance, meshes often arise from finite element or finite difference discretizations. In such cases, each mesh point is associated with a location in space. We can assign each graph vertex a set of coordinates that describe its spatial location. These

spatial locations can be used to partition data structures [3, 33].

Vertices may also be assigned weights to represent estimated computational costs. In order to accurately estimate the computational costs, we need information on how work will be partitioned. One way of deriving weights is to make the implicit assumption that an owner computes rule will be used to partition work. Under this assumption, computational cost associated with executing a statement will be attributed to the processor owning a left hand side array reference. The weight associated with a vertex in the loop L2 of Figure 11 would be proportional to the degree of the vertex, assuming functions f and g have identical computational costs. Vertex weights can be used as the sole partitioning criterion in problems in which computational costs dominate. Examples of such code include the flame simulation code described in Section 2.1.2 and "embarrassingly parallel problems", where computational cost predominates.

A given partitioner can make use of a combination of connectivity, geometrical and weight information. For instance, we find that it is sometimes important to take estimated computational costs into account when carrying out coordinate or inertial bisection for problems where computational costs vary greatly from node to node. Other partitioners make use of both geometrical and connectivity information [11].

Since the data structure that stores information on which data partitioning is to be based can represent **Geometrical**, **Connectivity** and/or **Load** information, we call this the **GeoCoL** data structure.

More formally, a GeoCoL graph $G = (V, E, W_v, W_e, C)$ consists of

1. a set of vertices $V = \{v_1, v_2, ..., v_n\}$, where $n = |V|$
2. a set of undirected edges $E = \{e_1, e_2, ..., e_m\}$, where $m = |E|$
3. a set of vertex weights $W_v = \{W_v^1, W_v^2, ..., W_v^n\}$
4. a set of edge weights $W_e = \{W_e^1, W_e^2, ..., W_e^m\}$, and
5. a set of coordinate information, for each vertex, of dimension d, $C = \{< c_1^1, ..., c_d^1 >, ..., < c_1^n, ..., c_d^n >\}$

4.1.2 Generating the GeoCoL Data Structure via a Compiler

We propose a directive **CONSTRUCT** that can be employed to direct a compiler to generate a GeoCoL data structure. A user can specify spatial information using the keyword **GEOMETRY**.

The following is an example of a GeoCoL declaration that specifies geometrical information:

C$ **CONSTRUCT** G1 (N, **GEOMETRY**(3, xcord, ycord, zcord))

This statement defines a GeoCoL data structure called **G1** having N vertices with spatial coordinate information specified by arrays xcord, ycord, and zcord. The GEOMETRY construct is closely related to the geometrical partitioning or *value based decomposition* directives proposed by von Hanxleden [17].

Similarly, a GeoCoL data structure that specifies only vertex weights can be constructed using the keyword **LOAD** as follows.

C$ **CONSTRUCT** G2 (N, **LOAD**(weight))

Here, a GeoCoL structure called **G2** consists of N vertices with vertex i having **LOAD** *weight*(i).

The following example illustrates how connectivity information is specified in a GeoCoL declaration. The Integer arrays **n1** and **n2** list the vertices associated with each of E graph edges and integer arrays **n1** and **n3** list vertices for another set of E edges.

C$ **CONSTRUCT** G3 (N, **LINK**(E, n1, n2), **LINK**(E, n1, n3))

The keyword **LINK** is used to specify the edges associated with the GeoCoL graph. The resultant edges of the GeoCoL data structure are the union of 1) edges linking *n1(i)* and *n2(i)* and 2) edges linking *n1(i)* and *n3(i)*.

Any combination of spatial, load and connectivity information can be used to generate GeoCoL data structure. For instance, the GeoCoL data structure for a partitioner that uses both geometry and connectivity information can be specified as follows:

C$ **CONSTRUCT** G4 (N, **GEOMETRY**(3, xcord, ycord, zcord), **LINK**(E, edge1, edge2))

Once the GeoCoL data structure is constructed, data partitioning is carried out. We assume there are **P** processors. At compile time *dependency coupling code* is generated. This code generates calls to the runtime support that, when the program executes:

1. generates the GeoCoL data structure
2. passes the GeoCoL data structure to a *data partitioning* procedure; the partitioner partitions the GeoCoL into **P** subgraphs, and
3. passes the new distribution information (the assignment of GeoCoL vertices to processors) to a runtime procedure to redistribute data

The GeoCoL data structure is constructed from the initial default distribution of the distributed arrays. Once we have the new distribution provided by the partitioner, we redistribute the arrays based on it. A communication schedule is built and used to redistribute the arrays from the default to the new distribution.

Vienna Fortran [43] provides support for the user to specify a function for distributing data. Within the function, the user can perform any processing to specify the data distribution.

4.2 Examples of Linking Data Partitioners

In Figure 12 we illustrate a possible set of partitioner coupling directives for the loop L1 in Figure 11. Statements S1 to S4 produce a default initial distribution of data arrays **x** and **y** and the indirection arrays **edge1** and **edge2** in loop L2. The statements S5 and S6 direct the generation of code to construct the GeoCoL graph and call the partitioner. Statement S5 indicates that the GeoCoL graph edges are to be generated based on the indirection arrays **edges1** and **edges2**. This information is provided by using the keyword **LINK** in the **CONSTRUCT** directive. The motivation for using the indirection arrays to construct the edges is that they represent the underlying data access patterns of the arrays **x** and **y** in loop L1. When the GeoCoL graph with edges representing the data access pattern is passed to the partitioner, the partitioner tries to break the graph into subgraphs such that the number of edges cut between the subgraphs is minimum. Hence, communication between processors is minimized. The statement S6 in the figure calls the partitioner RSB (recursive spectral bisection) with GeoCoL as input. The user is provided with a library of commonly available partitioners and

```
   REAL*8 x(nnode),y(nnode)
   INTEGER edge1(nedge), edge2(nedge)
S1 DYNAMIC, DECOMPOSITION reg(nnode), reg2(nedge)
S2 DISTRIBUTE reg(BLOCK), reg2(BLOCK)
S3 ALIGN x,y with reg
S4 ALIGN edge1, edge2 with reg2
   ....
   call read_data(edge1, edge2, ...)
S5 CONSTRUCT  G (nnode, LINK(nedge,edge1, edge2))
S6 SET distfmt BY PARTITIONING G USING RSB
S7 REDISTRIBUTE reg(distfmt)
C  Loop over edges involving x, y
L2 FORALL i = 1, nedge
   REDUCE (ADD, y(edge1(i)), f(x(edge1(i)), x(edge2(i))))
   REDUCE (ADD, y(edge2(i)), g(x(edge1(i)), x(edge2(i))))
   END FORALL

   ....
```

Figure 12: Example of Implicit Mapping in Fortran 90D

```
S5' CONSTRUCT G (nnode, GEOMETRY(3, xc, yc, zc))
S6' SET distfmt BY PARTITIONING G USING RCB

S7' REDISTRIBUTE reg(distfmt)
```

Figure 13: Example of Implicit Mapping using Geometry Information in Fortran 90D

can choose among them. Also, the user can link a customized partitioner as long as the calling sequence matches that of the partitioners in the library. Finally, the distributed arrays are remapped in statement S7 using the new distribution returned by the partitioner.

```
 S1 DYNAMIC, DECOMPOSITION grid(NPOINTS)
 S2 DISTRIBUTE grid(BLOCK)
 S3 ALIGN x(:), y(:), z(:), wt(:) WITH grid(BLOCK)
 K1 wt(1:NPOINTS) = 1
 K2 do J = 1, n_timestep
  C   Phase 1: Navier Stokes Solver - Convection Phase - BLOCK data distribution
 K3 FORALL i = 1, NPOINTS
 K4   x(i) = x(i) + F(y(i), y(i-1), y(i), y(i+1), z(i))
 K5 END FORALL
 S4 CONSTRUCT G (NPOINTS, LOAD(wt))
 S5 SET mydist BY PARTITIONING G USING BIN_PACKING
 S6 REDISTRIBUTE grid(mydist)
  C   Phase 2: Adaptive ODE Solver - Reaction Phase - IRREGULAR data distribution
 K6 wt(1:NPOINTS) = 1
 K7 FORALL i = 1, NPOINTS
 K8   z(i) = Adaptive_Solver(x(i),wt(i))
 K9 END FORALL
 S7 REDISTRIBUTE grid(BLOCK)
K10 end do
```

Figure 14: An Example of Adaptive Partitioning Using Fortran 90D

Figure 13 illustrates code similar to that shown in Figure 12 except that the use of geometric information is shown. Arrays **xc**, **yc**, and **zc**, which carry the spatial coordinates for elements in **x** and **y**, are aligned with the same decomposition to which arrays **x** and **y** are aligned. Statement S5' specifies that the GeoCoL data structure is to be constructed using geometric information. S6' specifies that recursive binary coordinate bisection (RCB) is used to partition the data.

Recall from Section 2.1.2 that the computation in the combustion code cycles over a convection phase and a reaction phase. The data access pattern in the convection phase involves accesses to only nearest neighbor array elements. Hence, during the convection phases it is reasonable to make use of a **BLOCK** distribution of data for arrays **x**, **y**, and **z**. Statements S1 through S2 in Figure 14 produce **BLOCK** distribution of data arrays. In the reaction phase, the amount of work done at each mesh point various as time progresses, and no communication occurs. The computational cost of the reaction phase at each mesh point in the current time step is stored in array **wt**. This cost information is used to distribute data arrays in the reaction phase of the next time step. A bin-packing heuristic is invoked to obtain the data distribution for the reaction phase. The statements S4 through S6 carry out the data distribution for the reaction phase.

4.3 Loop Iteration Partitioning

Once we have partitioned data, we must partition computational work. One convention is to compute a program assignment statement S in the processor that owns the distributed array element on S's left hand side. This

convention is normally referred to as the "owner-computes" rule. (If the left hand side of S references a replicated variable then the work is carried out in all processors). One drawback to the owner-computes rule in sparse codes is that we may need to generate communication within loops, even in the absence of loop carried dependencies. For example, consider the following loop:

FORALL i = 1,N

S1 x(ib(i)) = ...

S2 y(ia(i)) = x(ib(i))

END FORALL

This loop has a loop independent dependence between S1 and S2, but no loop carried dependencies. If we assign work using the owner-computes rule, for iteration i, statement S1 would be computed on the owner of ib(i) (OWNER(ib(i))) while statement S2 would be computed on the owner of ia(i) (OWNER(ia(i))). The value of y(ib(i)) would have to be communicated whenever OWNER(ib(i)) $\neq$ OWNER(ia(i)).

In Fortran D and Vienna Fortran, a user can specify on which processor to carry out a loop iteration using the **ON** clause. For example, in Fortran D, the above loop could be specified as

FORALL i = 1,N ON HOME(x(i))

S1 x(ib(i)) = ...

S2 y(ia(i)) = x(ib(i))

END FORALL

This means that iteration i must be computed on the processor on which x(i) resides (OWNER(x(i))), where the size of arrays **ia** and **ib** is equal to the number of iterations. Similar capabilities exist in Vienna Fortran.

When an **ON** clause is not explicitly specified, it is the compiler's responsibility to determine where to compute each iteration. An alternate policy to the owner computes rule is to assign all work associated with a loop iteration to a given processor. Our current default is to employ a scheme that executes a loop iteration on the processor that is the home of the largest number of distributed array references in an iteration, which we refer to as the "almost owner computes rule".

5 Runtime Support

In this section we briefly discuss the functionality of the runtime primitives that are used to perform the steps outlined in Figure 9. It should be noted that one of the important features of the approach taken in this work is the reliance upon an efficient runtime system.

The runtime support for compiler-embedded mapping presented in this paper can be broadly divided into three categories: 1) general support for communication and distributed data management, 2) data partitioning, and 3) iteration partitioning (work assignment). The following subsections briefly describe these primitives.

5.1 Data Partitioning

The runtime support associated with data partitioning includes procedures for generating distributed the GeoCoL data structure for partitioners (that operate on the GeoCoL data structure) to determine a data distribution, and for procedures to remap data as specified by the partitioner output.

The data structures describing the problem domain are specified by the "CONSTRUCT" directive discussed earlier. Processing this primitive requires generating a weighted interaction graph representing the computation load and/or communication dependencies. For example, the connectivity edges of the GeoCoL graph might reflect the read/write access patterns of the specified computation.

When connectivity information for the GeoCoL data structure is provided in the form of arrays (e.g. indirection arrays in an irregular loop), pre-processing is required to construct the connectivity graph. The procedures *eliminate_dup_edges* and *generate_geocol* could be used to do the pre-processing. Given the data access pattern information in the form of integer arrays **n1** and **n2**, the GeoCoL graph is constructed by adding an undirected edge $< n1(i), n2(i) >$ between nodes $n1(i)$ and $n2(i)$ of the graph.

Figure 15 shows the parallel generation of connectivity information in the GeoCoL data structure when integer indirection arrays are provided. Each processor generates local GeoCoL data structure using the local set of indirection arrays. The local graph is generated by the procedure *eliminate_dup_edges*. For clarity, the local GeoCoL is shown as an adjacency matrix. The local graphs are then merged to form a global distributed graph using the procedure *generate_geocol*. During the merge, if we view the local graph as an adjacency matrix stored in compressed sparse row format, processor P_0 collects all entries from the first N/P rows in the matrix from all other processors, where N is the number of nodes (array size) and P is the number of processors. Processor P_1 collects the next N/P rows of the matrix and so on. Processors remove duplicate entries when they collect adjacency list entries. The output of procedure *generate_geocol* is a GeoCoL data structure with the global connectivity information.

Any appropriate data partitioner may be used to compute the new data distribution using the GeoCoL graph. Table 2 lists many candidate partitioners for determining the data partitioning. In fact, a user may use any partitioner as long as the input and output data structures conform to those required by other primitives. The output of the partitioner describes a mapping of the data satisfying the desired criteria for load balance and communication minimization.

5.2 Workload Partitioning

Once data is partitioned, computation also must be partitioned. Workload (computation) partitioning refers to determining which processor will evaluate which expressions. Computation partitioning can be performed at several levels of granularity. At the finest level, each operation may be individually assigned to a processor. At the coarsest level, a block of iterations may be assigned to a processor, without considering the data distribution and access patterns. Both approaches seem expensive because, in the first case, the amount of preprocessing overhead can be very high, whereas in the second case communication cost can be very high. We have taken an approach which represents a compromise. We consider each loop iteration individually before assigning it to a processor.

For this purpose, we have developed data structures and run procedures to support iteration partitioning.

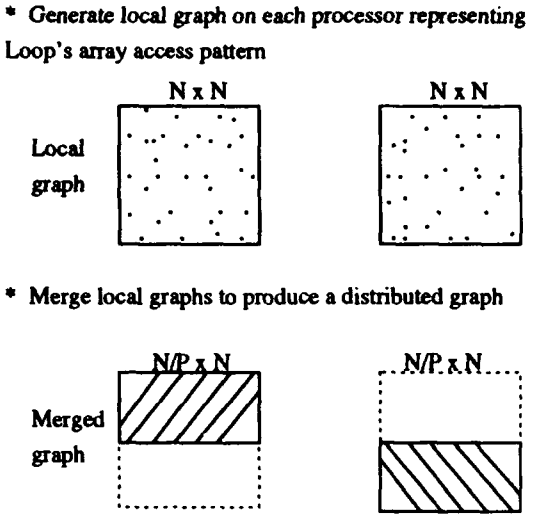

Figure 15: Parallel generation of GeoCoL graph

To partition loop iterations, we use a graph called the *runtime iteration graph*, or **RIG**. The **RIG** associates with each loop iteration i, all indices of each distributed array accessed during iteration i. A **RIG** is generated for every loop that references at least one irregularly distributed array.

Using the **RIG**, for each iteration we compute a list containing the number of distinct data references on each processor. Primitive *deref_rig* uses the **RIG** and the distributed translation tables to find the processor assignments associated with each distributed array reference. Subsequently, primitive *iteration_partitioner* uses this information to partition iterations. Currently, the heuristic used for iteration partitioning is the "almost owner computes" rule, in which an iteration is assigned to the processor which owns the majority of the elements participating in that particular iteration.

Note that just as there are many possible strategies that can be used to partition data, there are also many strategies that can be used to partition loop iterations. We are currently investigating techniques to specify "workload partitioners" or "iteration partitioners" in which a user can provide a customized heuristic.

5.3 Data Redistribution

For efficiency reasons, in scientific programs, distribution of distributed data arrays may have to be changed between computational domains or phases. For instance, as computation progresses in an adaptive problem, the work load and distributed array access patterns may change based on the nature of problem. This change might result in a poor load balance among processors. Hence, data must be redistributed periodically to maintain this balance.

To obtain an irregular data distribution for an irregular concurrent problem, we start with a known initial distribution δ_A of data arrays. Then, we apply a heuristic method to obtain an irregular distribution δ_B. Once we have the new data distribution, all data arrays associated with distribution δ_A must be transformed to distribution δ_B. For example, in solving the Euler equations of an unstructured grid, the flow variables are distributed in this method. Similarly, the loop iterations and the indirection arrays associated with the loop

Table 3: Performance of Schedule Reuse

(Time in Secs)	10K Mesh Processors		53k Mesh Processors		648 Atoms Processors		
	8	16	32	64	4	8	16
No Schedule Reuse	161	94	301	189	707	384	227
Schedule Reuse	10.1	8.4	19.8	17.0	15.2	9.7	8.0

must also be remapped.

To redistribute data and loop iteration space, we have developed a runtime procedure called *remap*. This procedure takes as input the original and the new distribution in the form of translation tables and returns a communication schedule. This schedule can be used to move data between initial and new distributions.

6 Experimental Results

This section presents the experimental results for the various techniques presented in this paper for compiler and runtime support for irregular problems. All measurements are performed on the Intel iPSC/860. In particular, we present the performance improvements obtained by employing communication schedule reuse, comparing the performance of compiler generated code with that of hand coded versions, and also present data on the performance of compiler-embedded mapping using various partitioners.

6.1 Communication Schedule Reuse

In this section, we present performance data for the schedule saving technique proposed in Section 3 for the Fortran 90D/HPF compiler implementation.

These performance measurements are for a loop over edges from 3-D unstructured Euler solver [31] for both 10K and 53K mesh points, and for an electrostatic force calculation loop in a molecular dynamics code for a 648 atom water simulation [6]. The functionality of these loops is equivalent to the loop L1 in Figure 11.

Table 3 presents the performance results of the compiler generated code with and without the schedule reuse technique. The table presents the execution time of the loops for 100 iterations with distributed arrays decomposed irregularly using a recursive coordinate bisection partitioner. Clearly, being able to reuse communication schedules improves performance tremendously. This is because without reuse, schedules must be regenerated at each time step , and therefore, the cost is proportional to the number of iterations.

6.2 Performance of the Mapper Coupler

In this section, we present performance results that compare the the costs incurred by the compiler generated mapper coupler procedures with the cost of a hand embedded partitioner.

To map arrays we employed two different kinds of parallel partitioners: (1) geometry based partitioners (coordinate bisection [3] and inertial bisection [33]), and (2) a connectivity based partitioner (recursive spectral

Table 4: Unstructured Mesh Template - 53K Mesh - 32 Processors

(Time in Secs)	Recursive Coordinate Bisection			Block Partition	
	Hand Coded	Compiler: No Schedule Reuse	Compiler Schedule Reuse	Hand Coded	Compiler Schedule Reuse
Partitioner	1.3	1.3	1.3	0.0	0.0
Inspector, remap	3.3	286	3.4	3.2	3.4
Executor	13.9	13.9	15.1	36.6	38.2
Total	18.5	301	19.8	39.8	41.6

Table 5: Unstructured Mesh Template - 53K Mesh - 32 Processors

(Time in Secs)	Inertial Bisection		Spectral Bisection	
	Hand Coded	Compiler Schedule Reuse	Hand Coded	Compiler: Schedule Reuse
Graph Generation	-	-	1.8	2.1
Partitioner	1.4	1.4	226	227
Inspector, remap	3.1	3.3	3.1	3.2
Executor	14.7	16.1	12.5	13.4
Total	19.2	20.8	243	246

bisection [37]). The performance of the compiler embedded mapper and a hand parallelized version are shown in Tables 4 and 5.

In Tables 4 and 5, *Partitioner* represents the time needed to partition the arrays using the partitioners, *Executor* depicts the time needed to carry out the actual computation and communication for 100 iterations (time steps), and *inspector and remap* show the time taken to build the communication schedule and redistribute data to the new distribution.

The Table 4 presents the performance of results of the Euler loop with the compiler-linked recursive coordinate bisection partitioner and the HPF BLOCK distribution for a 53K mesh template on 32 processors. Two important observations can be made from Table 4. First, the compiler generated code performs almost as well

Table 6: Performance of Compiler-linked Coordinate Bisection Partitioner with Schedule Reuse

Tasks (Time in Secs)	10K Mesh		53k Mesh		648 Atoms		
	Processors		Processors		Processors		
	8	16	32	64	4	8	16
Partitioner	0.4	0.4	1.3	3.1	0.1	0.1	0.1
Inspector	0.4	0.3	0.9	0.5	2.2	1.2	0.7
Remap	1.4	0.9	2.5	1.7	4.8	2.6	1.5
Executor	7.9	6.9	15.1	11.7	8.1	5.8	5.7
Total	10.1	8.5	19.8	17.0	15.2	9.7	8.0

Table 7: Compiler Performance for Block Distribution with Schedule Reuse

Tasks (Time in Secs)	10K Mesh Processors		53k Mesh Processors		648 Atoms Processors		
	8	16	32	64	4	8	16
Inspector	0.5	0.3	1.2	0.7	2.7	1.5	0.8
Remap	1.2	0.8	2.2	1.0	4.5	2.6	1.5
Executor	17.6	12.6	38.2	28.6	10.3	7.6	7.3
Total	19.3	13.7	41.6	30.3	17.5	11.7	9.6

as the hand written code. In fact, the compiler generated code is within 15% of the hand coded version. The overhead is partly due to book-keeping done to reuse schedules and partly due to runtime calculation of loop bounds. Second, the performance of the code using the partitioner is much better than the performance of the block partitioned code even when the cost of executing the partitioner is included.

Table 5 shows the performance of compiler generated code when two additional partitioners are used; namely, recursive spectral bisection (RSB) and inertial bisection. In Table 5, *Partitioner* for Spectral Bisection depicts the time needed to partition the GeoCoL graph data structure using a parallelized version of Simon's single level spectral partitioner [37]. Only a modest effort was made to produce an efficient parallel implementation of the partitioner and we believe that the performance and the execution time of the partitioner can be tremendously improved by using a multilevel version of the partitioner [2, 21]. We partitioned the GeoCoL graph into a number of subgraphs equal to the number of processors employed. It should be noted that any parallelized partitioner could be used. The *graph generation* time depicts the time required to generate GeoCoL graph.

Clearly, different partitioners perform differently in terms of execution time and the quality of load balancing. We observe that the best load balancing is obtained by using RSB because the time for the executor phase is minimized. However, the cost of partitioning using RSB is quite high. Thus, the choice of a partitioner should depend on how long the solution of a problem is likely to take (the number of time steps).

Table 6 shows the performance of the compiler generated code for the Euler and the molecular dynamics loops on various number of processors. To compare the partitioner's performance for different programs, we have also included timings for a hand coded block partitioned version in Table 7. In the blocked version, we assigned each contiguous blocks of array elements to processors using the HPF BLOCK distribution. The use of either a coordinate bisection partitioner or a spectral bisection partitioner led to a reduction factor of two to three improvement in the executor time compared to the use of block partitioning. This example also points out the importance of the number of executor iterations and choice of partitioner. When compared to the recursive coordinate bisection partitioner, the recursive spectral bisection partitioner is associated with faster time per executor iteration but also a significantly higher partitioning overhead. Irregular distribution of arrays performs significantly better than the existing BLOCK distribution supported by HPF.

6.3 Performance of Adaptive Problems

We now present experimental results for an application of the type described in Section 2.1.2. Recall that

Table 8: Performance of Combustion Code With Compiler-Linked Load based Partitioner

Grid Size (Time in Sec.)	Processors	Hand Coded			Compiler Generated			No Load Balance
		Load Balance	Comp	Total	Load Balance	Comp	Total	Total
1024x32	16	4.1	19.4	23.5	4.3	19.4	23.7	117
	32	2.8	12.4	15.2	2.9	12.8	15.8	61
1024x128	32	8.5	34.1	42.9	8.5	34.6	43.4	397
	64	5.2	26.4	31.6	5.2	26.9	32.1	342

this type of application alternates between two distinct computational phases. The first phase (*convection*) consists of structured calculations on a Cartesian mesh. The second phase (*reaction*) involves a set of local computations at each mesh point. The computational cost associated with the reaction phase varies between mesh points. Figure 6 in Section 2.1.2 depicts the computational structure of this type of application.

The results we present are for a simplified version of the Reactive Euler solver developed by James Weber at the University of Maryland. This algorithm computes the reaction rates of various gases, integrates the governing rate equations, and determines the new *number densities* in an hypersonic medium. The thermodynamic quantities, such as temperature, pressure, and specific heat ratio are evaluated as the reaction mechanism proceeds. The first phase of the Reactive Euler solver is an explicit Navier Stokes solver, while the second phase is an adaptive ordinary differential equation solver.

Figure 14 depicts our load balancing strategy. In this simplified example, we represent the mesh as a one dimensional array. The array is partitioned into equal-size blocks (i.e. a **BLOCK** mapping). In order to ensure a good load balance during the reaction phase, we redistribute only expensive reaction calculations. Reaction calculations are redistributed based on the costs incurred in the previous time step. After the reaction phase, the remapped data are returned to their original positions.

Table 8 presents the performance of the second reaction phase for 100 cycles, and a comparison between hand coded and compiler generated code. The *Load Balance* columns give the time taken to carry out the partitioner and remap the data. We used a bin-packing heuristic to balance the load in the combustion phase. We observe that the performance of the compiler generated code is almost as good as that of the hand coded version. Also note the performance improvements obtained when using a load based partitioner and adaptivity compared to performing no load balancing.

Finally, Table 9 summarizes the compiler performance for all the codes and presents a comparison with the hand coded version. For all problems, the performance of the compiler generated code is within 15% of that of the hand coded version.

7 Related Work

Research has been carried out by von Hanxleden [17] on compiler-linked partitioners that decompose arrays based on distributed array element values; these are called *value based decompositions*. Our GEOMETRY construct can be viewed as a particular type of value based decomposition. Several researchers have developed programming environments that are targeted toward particular classes of irregular or adaptive problems. Williams [41]

Table 9: Performance of Compiler-linked Partitioners

Total Time (in Secs)	10K Mesh		53k Mesh		648 Atoms			1024x32 Grid	
	Processors		Processors		Processors			Processors	
	8	16	32	64	4	8	16	16	32
Hand coded	8.8	7.0	18.5	14.9	14.3	8.5	7.0	23.5	15.2
Compiler	10.1	8.5	19.8	17.0	15.2	9.7	8.0	23.7	15.8

describes a programming environment (DIME) for calculations with unstructured triangular meshes using distributed memory machines. Baden [1] has developed a programming environment targeted towards particle computations. This programming environment provides facilities that support dynamic load balancing.

There are a variety of compiler projects targeted at distributed memory multiprocessors: the Fortran D compiler projects at Rice and Syracuse [15, 5] and the Vienna Fortran compiler project [43] at the University of Vienna are two examples. The Jade project at Stanford [26], the DINO project at Colorado [35], Kathy Yelick's work [7] at Berkeley, and the CODE project at UT, Austin provide parallel programming environments. Runtime compilation methods have been employed in four compiler projects: the Fortran D project [22], the Kali project [25], Marina Chen's work at Yale [28] and the PARTI project [32, 42, 36]. The Kali compiler was the first compiler to implement inspector/executor type runtime preprocessing [25] and the ARF compiler was the first compiler to support irregularly distributed arrays [42].

In earlier work, we have outlined a strategy (but did not attempt a compiler implementation) that would make it possible for compilers to generate compiler embedded connectivity based partitioners directly from marked loops [12]. The approach described here requires more input from the user and less compiler support.

8 Conclusions

We have described work that demonstrates two new mechanisms for dealing effectively with irregular computations. The first mechanism invokes a user specified mapping procedure using a set of compiler directives. The second mechanism is a simple conservative method that in many cases makes it possible for a compiler to recognize the potential for reusing previously computed results from inspectors (e.g. communication schedules, loop iteration partitions, information that associates off-processor data copies with on-processor buffer locations).

We view the CHAOS procedures described here as forming a portion of a portable, compiler independent, runtime support library. The CHAOS runtime support library contains procedures that

1. support static and dynamic distributed array partitioning
2. partitions loop iterations and indirection arrays
3. remap arrays from one distribution to another and
4. carry out index translation, buffer allocation and communication schedule generation

We tested our prototype compiler on computational templates extracted from an unstructured mesh computational fluid dynamics code, a molecular dynamics code and an hypersonic combustion code. We embedded our

runtime support by hand and compared its performance against the compiler generated code. The compiler's performance on these templates was within 15% of the hand compiled code.

In our current implementation, we have performed iteration partitioning of a Forall loop using the almost owner computes rule. In general, for data partitioning, a user or compiler should be able to specify a partitioner to perform iteration partitioning. We are currently developing primitives to couple iteration partitioners with Fortran 90 Forall loops.

The CHAOS procedures described in this paper are available for public distribution and can be obtained from netlib or from the anonymous ftp site hyena.cs.umd.edu.

Acknowledgments

The authors would like to thank Alan Sussman and Raja Das for many fruitful discussions; Shamik Sharma and Mustafa Uysal for their part in constructing the CHAOS runtime support; Donna Meisel for careful proof reading of this manuscript.

The authors would like to thank Chuck Koelbel and Sanjay Ranka for many enlightening discussions about universally applicable partitioners and how to embed such partitioners in compilers; we would also like to thank Chuck Koelbel, Ken Kennedy and Seema Hiranandani for many useful discussions about integrating Fortran-D runtime support for irregular problems. Special thanks go to Reinhard von Hanxleden for his helpful suggestions.

The authors would also like to gratefully acknowledge Zeki Bozkus and Tom Haupt for the time they spent explaining the internals of the Fortran 90D compiler. We would also like to thank Horst Simon for the use of his unstructured mesh partitioning software.

References

[1] S. Baden. Programming abstractions for dynamically partitioning and coordinating localized scientific calculations running on multiprocessors. *SIAM J. Sci. and Stat. Computation.*, 12(1), January 1991.

[2] S.T. Barnard and H. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. Technical Report RNR-92-033, NAS Systems Division, NASA Ames Research Center, November 1992.

[3] M.J. Berger and S. H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. on Computers*, C-36(5):570–580, May 1987.

[4] Harry Berryman, Joel Saltz, and Jeffrey Scroggs. Execution time support for adaptive scientific algorithms on distributed memory machines. *Concurrency: Practice and Experience*, 3(3):159–178, June 1991.

[5] Zeki Bozkus, Alok Choudhary, Geoffrey Fox, Tomasz Haupt, Sanjay Ranka, and Min-You Wu. Compiling Fortran 90D/HPF for distributed memory MIMD computers. To appear in the Journal of Parallel and Distributed Computing, March 1993.

[6] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. Charmm: A program for macromolecular energy, minimization, and dynamics calculations. *Journal of Computational Chemistry*, 4:187, 1983.

[7] Soumen Chakrabarti and Katherine Yelick. Implementing an irregular application on a distributed memory multiprocessor. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, May 1993. ACM SIGPLAN Notices, Vol. 28, No. 7.

[8] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, Fall 1992.

[9] Barbara Chapman, Piyush Mehrotra, and Hans Zima. Programming in Vienna Fortran. Technical Report 92-9, ICASE, NASA Langley Research Center, March 1992.

[10] A. Choudhary, G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, S. Ranka, and J. Saltz. Software support for irregular and loosely synchronous problems. *Computing Systems in Engineering*, 3(1-4):43–52, 1992. Papers presented at the Symposium on High-Performance Computing for Flight Vehicles, December 1992.

[11] T. W. Clark, R. v. Hanxleden, J. A. McCammon, and L. R. Scott. Parallelization strategies for a molecular dynamics program. In *Intel Supercomputer University Partners Conference*, Timberline Lodge, Mt. Hood, OR, April 1992.

[12] R. Das, R. Ponnusamy, J. Saltz, and D. Mavriplis. Distributed memory compiler methods for irregular problems - data copy reuse and runtime partitioning. In *Compilers and Runtime Software for Scalable Multiprocessors, J. Saltz and P. Mehrotra Editors*, Amsterdam, The Netherlands, 1992. Elsevier.

[13] R. Das and J. H. Saltz. Program slicing techniques for compiling irregular problems. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland , OR, August 1993.

[14] D. Loveman (Ed.). Draft High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, January 1993.

[15] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Department of Computer Science Rice COMP TR90-141, Rice University, December 1990.

[16] S. Hammond and T. Barth. An optimal massively parallel Euler solver for unstructured grids. *AIAA Journal, AIAA Paper 91-0441*, January 1991.

[17] R. v. Hanxleden. Compiler support for machine independent parallelization of irregular problems. Technical Report CRPC-TR92301-S, Center for Research on Parallel Computation, Rice University, November 1992. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR92301-S`.

[18] R. v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. In *Proceedings of the 5th Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[19] R. v. Hanxleden, K. Kennedy, and J. Saltz. Value-based distributions in Fortran D — a preliminary report. Technical Report CRPC-TR93365-S, Center for Research on Parallel Computation, December 1993.

[20] R. v. Hanxleden and L. R. Scott. Load balancing on message passing architectures. *Journal of Parallel and Distributed Computing*, 13:312–324, 1991.

[21] B. Hendrickson and R. Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. Technical Report SAND 92-1460, Sandia National Laboratory, Albuquerque, September 1992.

[22] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. In *Compilers and Runtime Software for Scalable Multiprocessors, J. Saltz and P. Mehrotra Editors*, Amsterdam, The Netherlands, To appear 1991. Elsevier.

[23] A. Jameson, T. J. Baker, and N. P. Weatherhill. Calculation of inviscid transonic flow over a complete aircraft. *AIAA paper 86-0103*, January 1986.

[24] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, February 1970.

[25] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory architectures. In *2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 177–186. ACM, March 1990.

[26] Monica Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of block algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 63–74. ACM Press, April 1991.

[27] W. E. Leland. Load-balancing heuristics and process behavior. In *Proceedings of Performance 86 and ACM SIGMETRICS 86*, pages 54–69, 1986.

[28] L. C. Lu and M.C. Chen. Parallelizing loops with indirect array references or pointers. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[29] N. Mansour. Physical optimization algorithms for mapping data to distributed-memory multiprocessors. Technical report, Ph.D. Dissertation, School of Computer Science,Syracuse University, 1992.

[30] D. J. Mavriplis. Adaptive mesh generation for viscous flows using delaunay triangulation. *Journal of Computational Physics*, 90(2):271–291, 1990.

[31] D. J. Mavriplis. Three dimensional unstructured multigrid for the Euler equations, paper 91-i549cp. In *AIAA 10th Computational Fluid Dynamics Conference*, June 1991.

[32] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nicol, and Kay Crowley. Principles of runtime support for parallel processors. In *Proceedings of the 1988 ACM International Conference on Supercomputing*, pages 140–152, July 1988.

[33] B. Nour-Omid, A. Raefsky, and G. Lyzenga. Solving finite element equations on concurrent computers. In *Proc. of Symposium on Parallel Computations and theis Impact on Mechanics*, Boston, December 1987.

[34] G. Patnaik, K.J. Laskey, K. Kailasanath, E.S. Oran, and T. V. Brun. FLIC – A detailed, two-dimensional flame model. NRL Report 6555, Naval Research Laboratory, Washington, DC, September 1989.

[35] Matthew Rosing, Robert B. Schnabel, and Robert P. Weaver. The DINO parallel programming language. *Journal of Parallel and Distributed Computing*, 13(1):30–42, September 1991.

[36] J. Saltz, H. Berryman, and J. Wu. Runtime compilation for multiprocessors. *Concurrency: Practice and Experience*, 3(6):573–592, 1991.

[37] H. Simon. Partitioning of unstructured mesh problems for parallel processing. In *Proceedings of the Conference on Parallel Methods on Large Scale Structural Analysis and Physics Applications*. Pergamon Press, 1991.

[38] V. Venkatakrishnan, H. D. Simon, and T. J. Barth. A MIMD implementation of a parallel Euler solver for unstructured grids, submitted to Journal of Supercomputing. Report RNR-91-024, NAS Systems Division, NASA Ames Research Center, Sept 1991.

[39] P. Venkatkrishnan, J. Saltz, and D. Mavriplis. Parallel preconditioned iterative methods for the compressible navier stokes equations. In *12th International Conference on Numerical Methods in Fluid Dynamics, Oxford, England*, July 1990.

[40] N. P. Weatherill. The generation of unstructured grids using dirichlet tessalations. Report MAE 1715, Princeton, July 1985.

[41] R. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency, Practice and Experience*, 3(5):457–482, February 1991.

[42] J. Wu, J. Saltz, S. Hiranandani, and H. Berryman. Runtime compilation methods for multicomputers. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume 2, pages 26–30, 1991.

[43] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran – a language specification. Report ACPC-TR92-4, Austrian Center for Parallel Computation, University of Vienna, Vienna, Austria, 1992.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY(Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1993	Contractor Report

4. TITLE AND SUBTITLE
RUNTIME SUPPORT AND COMPILATION METHODS FOR USER-SPECIFIED DATE DISTRIBUTIONS

5. FUNDING NUMBERS
C NAS1-19480
WU 505-90-52-01

6. AUTHOR(S)
Ravi Ponnusamy, Joel Saltz, Alok Choudhury, Yuan-Shin Hwang, and Geoffrey Fox

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Computer Applications in Science
and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER
ICASE Report No. 93-99

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CR-191587
ICASE Report No. 93-99

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Michael F. Card
Final Report
To be submitted to IEEE Transactions on Parallel and Distributed Computing

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified–Unlimited

Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
This paper describes two new ideas by which an HPF compiler can deal with irregular computations effectively. The first mechanism invokes a user specified mapping procedure via a set of compiler directives. The directives allow use of program arrays to describe graph connectivity, spatial location of array elements and computational load. The second mechanism is a simple conservative method that in many cases enables a compiler to recognize that it is possible to reuse previously computed information from inspectors (e.g. communication schedules, loop iteration partitions, information that associates off-processor data copies with on-processor buffer locations). We present performance results for these mechanisms from a Fortran 90D compiler implementation.

14. SUBJECT TERMS	15. NUMBER OF PAGES
runtime optimizations; HPF compilers; data distributions; unstructured applications	35
	16. PRICE CODE A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		

NSN 7540-01-280-5500

Standard Form 298(Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

☆U.S. GOVERNMENT PRINTING OFFICE: 1994 - 528-064/86109